Criptografia, Descriptografia e fatoração do sistema RSA

THIAGO BRANDENBURG, Universidade do Estado de Santa Catarina, Brasil
ANA VEDOY, Universidade do Estado de Santa Catarina, Brasil

Este relatório é parte do trabalho final da disciplina de Complexidade de Algoritmos e consiste na implementação do criptografia e quebra por fatoração do algoritmo rsa. A linguagem de programação utilizada foi Haskell, conforme os gráficos percebe-se que a criptografia, descriptografia e geração de chave ocorrem em tempo polinomial, enquanto a fatoração ocorre em tempo exponencial.

CCS Concepts: • **Theory of computation** → Complexity theory and logic.

Additional Key Words and Phrases: criptografia,RSA,complexidade,fatoração

ACM Reference Format:
Thiago Brandenburg and Ana Vedoy. 2022. Criptografia, Descriptografia e fatoração do sistema RSA. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (July 2022), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUÇÃO

O Rivest-Shamir-Adleman (RSA) é um sistema de criptografia de chave pública, que se utiliza da não existência de algoritmos eficientes para fatoração de inteiros grandes, para criar um sistema teoricamente quebrável, mas inviável devido a complexidade de tempo exponencial. O RSA consiste na seleção de dois primos aleatórios (geralmente definidos como p e q), multiplicando-os para a geração do valor de chave pública (tipicamente chamada de n). O parâmetro auxiliar da chave pública (e) é um primo relativo arbitrário de φ= (p-1)*(q-1), e o parâmetro auxiliar da chave privada (d) é o inverso modular de *e* na base φ, calculado usando o algoritmo de Euclides Estendido. Este trabalho consiste na analise da implementação do sistema RSA e da fatoração da chave pública, com testes de tempo de execução para gerar chaves, encriptar, descriptar

2 IMPLEMENTAÇÃO

A linguagem de programação utilizada foi Haskell, a qual é especialmente útil para esse tipo de implementação pois Inteiros Grandes são nativamente aceitos na linguagem. Por ser uma linguagem funcional, não é necessário implementação de fluxos de controle robustos, na implementação, por exemplo, não foi necessário manipular bits explicitamente, pois as funções padrões oferecidas pelo Haskell já são sobrecarregadas e otimizadas e adapta para lidar com Inteiros Grandes.

A implementação foi dividida em 5 partes: implementação das funções para a geração de números randômicos, funções para geração dos primos, funções da criptografia RSA e funções de execução principal

Authors' addresses: Thiago Brandenburg, Universidade do Estado de Santa Catarina, Joinville, Brasil, thiagobranden@hotmail.com; Ana Vedoy, Universidade do Estado de Santa Catarina, Joinville, Brasil, vedoyalves@gmail.com.

2476-1249/2022/7-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

2.1 Sistema RSA

O RSA foi implementado de forma a retornar uma tupla contendo a chave privada e pública, como demonstrado no algoritmo 1, para criptografar ou descriptografar um número no sistema RSA é necessário apenas aplicar a Potência Modular, tendo isso em conta tal operação foi realizada em funções de encriptação e decriptação para facilitar o entendimento do código.

Listing 1. Geração de Chave Pública e Privada RSA

```
rsa_chave :: (Integer,Integer) -> ((Integer,Integer),(Integer,Integer))
rsa_chave (p,q) =
    let
        n = p*q
        phi = (p-1)*(q-1)
        e = aux 3 phi
        d = if fator < 0 then fator+2*phi else fator
        (fator,_) = euclides_ext e phi
        aux k phi =
            if gcd k phi > 1 then aux (k+2) phi
            else k
    in
        ((n,e),(n,d))
```

3 ANÁLISE DE COMPLEXIDADE

Para este relatório foram feitos a analise de complexidade da criptografia RSA, do teste de primalidade e da fatoração por força bruta da chave pública do RSA.

4 CRIPTOGRAFIA E DESCRIPTOGRAFIA

A Criptografia e descriptografia de uma mensagem m no sistema RSA consiste em aplicar a potencia modular em m para obter a mensagem cifrada C, sendo a potencia e para encriptar e d para decriptar, e o modulo aplicado n, conforme a equações 1 e 2

$$C = m^e mod n \tag{1}$$

$$m = C^d mod n(\text{descriptografia}) \tag{2}$$

O método escolhido para realizar a exponenciação modular foi o método binário, no qual a operação é realizada para cada casa binária do expoente, conforme o algoritmo 2

Listing 2. Potência modular

```
modPow :: Integer -> Integer -> Integer -> Integer -> Integer
modPow b e 1 r = 0
modPow b 0 m r = r
modPow b e m r
  | e `mod` 2 == 1 = modPow b' e' m (r * b `mod` m)
  | otherwise = modPow b' e' m r
  where
```

```
    b' = b * b `mod` m
    e' = e `div` 2
```

Considerando que a base, o expoente e o módulo são números inteiros grandes de n bits, o método binário realiza uma multiplicação binária para cada bit do expoente, como a complexidade da multiplicação é $O(n^2)$ sendo n o número de bits, então a complexidade do método binário é $O(n^3)$, onde n é o numero de bits.

4.1 Teste de Primalidade

Para a geração de Primos, foi utilizado o teste de primalidade de Miller-Rabin, um algoritmo probabilístico que retorna se um número é provavelmente primo. A probabilidade de erro do teste é de $<4^{-k}$, sendo k o número de repetições do teste. A implementação do método é observada no algoritmo 3. O teste foi utilizado para verificar números ímpares consecutivos, assim podendo tanto listar primos em um intervalo ou determinar qual é o próximo primo maior que um número qualquer.

Listing 3. Teste de primalidade de Miller-Rabin

```
m_r_primalidade :: Integer -> Int -> Bool
m_r_primalidade _ 0 =
    True
m_r_primalidade n k =
    let
        (s,d) = fatoracaoPorDois n
        (a,_) = randomR (2,n-2) (mkStdGen k)
        x = modPow a d n 1;
    in
        if x ==1 || x == (n-1) then m_r_primalidade n (k-1)
        else teste x n (s-1) k
        where
            teste :: Integer -> Integer -> Integer -> Int -> Bool
            teste x n 0 k = False
            teste x n s k =
                let x1 = modPow x 2 n 1 in
                if x1 == n-1 then m_r_primalidade n (k-1)
                else teste x1 n (s-1) k
```

O teste de primalidade de miller rabin para um número n é executado um k vezes, sendo k um número arbitrário que afeta a precisão. Para cada iteração de k, é realizado um outro loop s-1 vezes, onde s provem de $n = 2^s d + 1$, no pior caso (d=1) s é igual ao número de bits de n (chamaremos de α). No loop interno é realizado uma potência modular $x = x^2 \bmod n$, sabemos que $x^2 \bmod n = (x*x) \bmod n$, tanto da multiplicação quando do modulo para Inteiros Grandes é $O(\alpha^2)$, portanto a complexidade do teste de miller rabin é $O(k\alpha^3)$, sendo k é o numero de testes realizado e α o número de bits do número testado.

4.2 Fatoração da Chave pública por força bruta

O conceito da fatoração por força bruta é simples: n=p*q, p e q são primos, portanto p e q são o resultado da fatoração de n, então a fatoração da chave pública consiste em buscar por força bruta testar pares de números menores que n. Existem algumas otimizações que podem ser feitas, a primeira é que sabemos que todo primo

maior que 2 é ímpar, portanto podemos pular todos os números pares do intervalo. Poderíamos também usar o teste de primalidade para testar somente primos, no entanto, o teste de primalidade de miller rabin possui complexidade maior do que a multiplicação de inteiros grandes, então é mais produtivo multiplicar todos os ímpares do intervalo. A implementação utilizada segue no algoritmo

Listing 4. fatoração por força bruta da chave pública

```
rsa_forca_bruta :: (Integer,Integer) -> (Integer,Integer)
rsa_forca_bruta (n,e) = rsa_fat n (3,n)

rsa_fat :: Integer -> (Integer,Integer) -> (Integer,Integer)
rsa_fat n (a,b) =
  if mod a 2 ==0 then rsa_fat n (a+1,b) else
  if mod b 2 ==0 then rsa_fat n (a,b-1) else
  aux n (a,b)
  where
      aux n (a,b) =
        if a > b then (0,0) else
        let nx = a*b in if nx == n then (a,b) else
            if nx > n then aux n (a,b-2) else aux n (a+2,b)
```

Sendo α o número de bits de n, existem na ordem de 2^{a-1} número ímpares menores que n, e para cada par desses números é realizado uma multiplicação que involve inteiro grandes, com complexidade $O(\alpha^2)$, portanto a complexidade da força bruta é $O(2^a a^2)$, onde α é o número de bits de n.

5 TESTES REALIZADOS

Foram realizados testes de tempo para a geração de chaves, criptografia, descriptografia e fatoração da chave pública.

5.1 Geração de Chave

A geração de chaves é polinomial, pois apenas executa em sequencialmente operações de potência modular e aplica o algoritmo de Euclides Estendido, que também é polinomial. A curva polinomial pode ser observada na figura 1 à seguir.

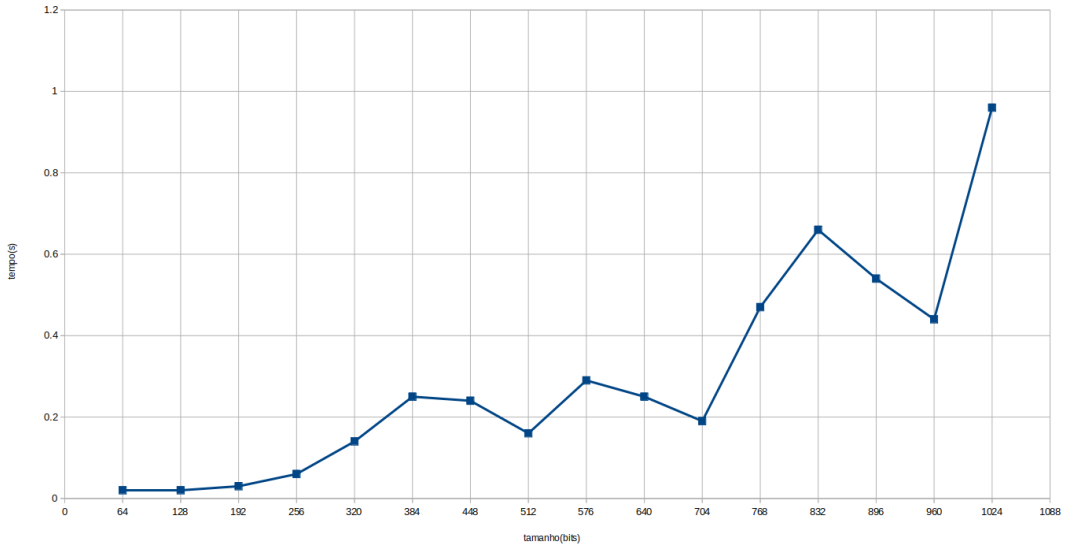

Fig. 1. Tempo para geração de chaves de 64 à 1024 bits

5.2 Criptografia e Descriptografia do Texto

A criptografia e descriptografia são processos também polinomiais, entretanto, o processo é tão otimizado (utilizando o método binário) que precisa-se de números extremamente robustos para observarmos a curva do algoritmo.

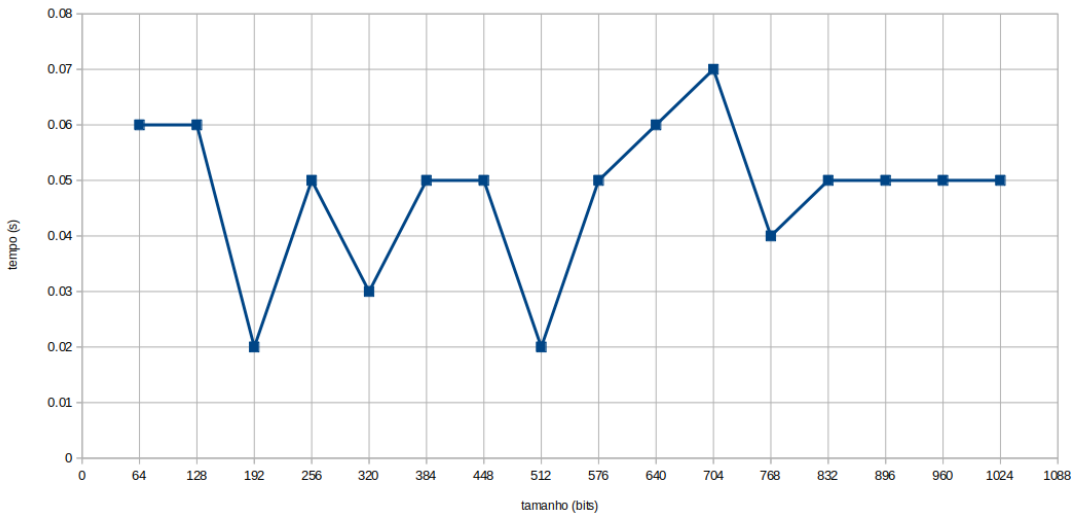

Fig. 2. Encriptação de valores com chaves de 64 à 1024 bits

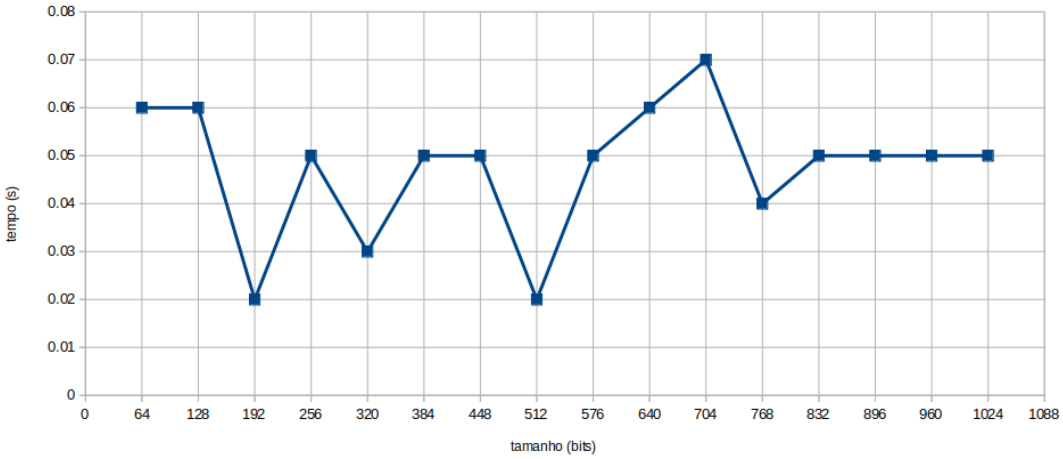

Fig. 3. Descriptação de valores com chaves de 64 à 1024 bits

5.3 Fatoração da Chave Pública

A fatoração da chave pública é o centro da segurança da criptografia RSA, o processo é polinomial, sem algoritmos muito bem otimizados e paralelos, dificilmente se passa de valores acima de 30 bits. A maior fatoração já feita de um algoritmo RSA foi 250 bits, realizada em 2020, com 2700 anos-core de processamento.

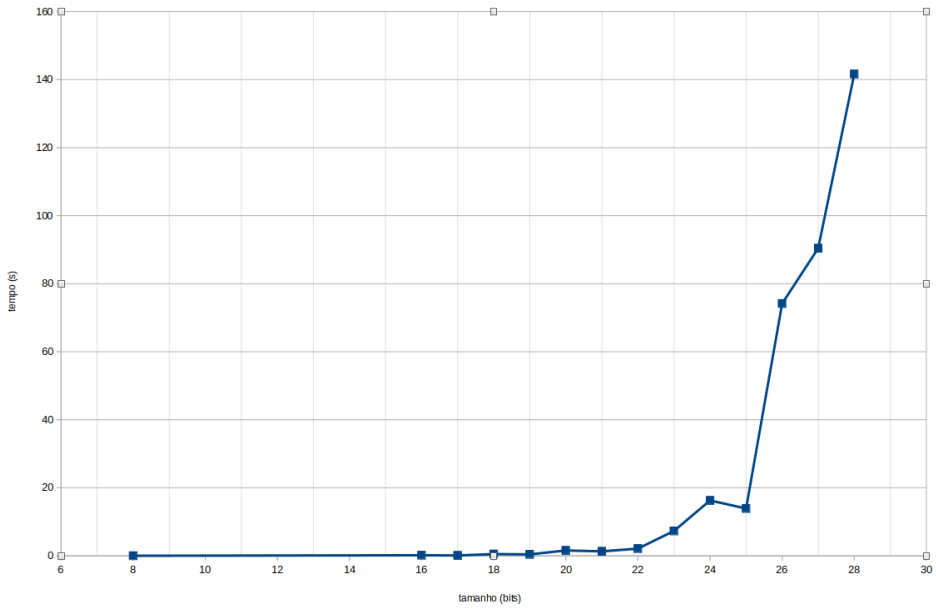

Fig. 4. Quebra por fatoração